Steering Massively Parallel Applications Under Python

Patrick Miller
Lawrence Livermore National Laboratory
patmiller@llnl.gov

This work was performed under the auspices of the U.S. Department of Energy by University of California, Lawrence Livermore National Laboratory under Contract W-7405-Eng-48.

Characterization of Scientific Simulation Codes

- Big!
  - Many options
  - Many developers
- Long Lived
  - Years to develop
  - 20+ year life
- Used as a *tool* for understanding
  - Used in unanticipated ways

Steering

- Steering uses an interpreted language as the interface to “compiled assets.”
  - E.g. Mathematica
- Rich command structure lets users “program” within simulation paradigm
- Exploit interactivity

Why Steering

- Expect the unexpected
- Let end-users participate in solutions to their problems
- Users can set, calculate with, and control low level details
- Developers can test and debug at a higher level
- Users can write their own instrumentation or even full blown physics packages
- It works!
  - Mathematica, SCIrun, BASIS, IDL, Tecolote (LANL), Perl, Python!!!!

Why Python?

- Objects everywhere
- Loose enough to be practical
- Small enough to be safe
  - The tiny core could be maintained by one person
- Open source for the core AND many useful bits & pieces already exist

The big picture

- C++ and FORTRAN compiled assets for speed
- Python for flexibility
- Small, nimble objects are better than monolithic objects
- No "main" - The control loop(s) are in Python
  - Python co-ordinates the actions of C++ objects
  - Python is the integration “glue”

Putting it together - Pyffle

- The biggest issue is always getting C++ to "talk with" Python (wrapping)
- SWIG, PyFORT, CXX Python, Boost Python, PYFFLE
- Our big requirements are...
  - support for inheritance
  - support for Python-like interfaces
  - tolerance of templating<>

The shadow layer

- Pyffle generates constructor functions - NOT Python classes
- We build extension classes in Python that "shadow" the C++ object to
  - Allow Python style inheritance
  - Add functionality
  - Enforce consistency
  - Modify interface
  - ...

Parallelism

- Kull uses SPMD style computation with a MPI enabled Python co-ordinating
  - *Most* parallelism is directly controlled "under the covers" by C++ (MPI and/or threads)
  - Started with a version with limited MPI support (barrier, allreduce)
  - New pyMPI supports a majority of MPI standard calls (comm, bcast, reduces, sync & async send/recv, ...)

Basic MPI in Python

- mpi.bcast(value)
- mpi.reduce(item,operation, root?)
- mpi.barrier()
- mpi.rank and mpi.nprocs
- mpi.send(value,destination, tag?)
- mpi.recv(sender,tag?)
- mpi.sendrecv(msg, dest, src?, tag?, rtag?)

MPI calcPi

```
def f(a):  return 4.0/(1.0 + a*a)
def integrate(rectangles,function):
        n = mpi.bcast(rectangles)
        h = 1.0/n
        for i in range(mpi.rank+1, n+1, mpi.procs):
                sum = sum + function( h * (i-0.05))
        myAnswer = h * sum
        answer = mpi.reduce(myAnswer, mpi.SUM)
        return answer
```

Making it fast

- Where we have very generic base classes in the code (e.g. Equation-of-state), we have written Pythonized descendant classes that invoke arbitrary user written Python functions
- C++ component doesn't need to know it's invoking Python
- There is a speed issue :-(
- But there is hope!

PyCOD - Compile on demand!

- Builds accelerated Python functions AND C++ functions
- Input is a Python function object and a type signature to compile to
- Dynamically loads accellerated versions
- Caches compiled code (matched against bytecode) to eliminate redundant compiles
- Early results are encouraging (20X to 50X speedup)

PyCOD example

```
import compiler
from types import *

def sumRange(first,last):
  sum = 0
  for i in xrange(first,last+1):
    sum = sum + i
  return sum


signature = (IntType,IntType,IntType
compiledSumRange, rawFunctionPointer = \
      compiler.accelerate(sumRange,signature)

print "Compiled result:", compiledSumRange(1,100)
```

Summary

- MPI enabled Python is a reality
  - We have run 1500 processors, long running simulations running fully under a Python core
- PYFFLE provides a reasonable conduit for developers to build capability in C++ and easily integrate that capability into Python